Evolving Multi-tenant SaaS Applications through Self-Adaptive Upgrade Enactment and Tenant Mediation

Fatih Gey, Dimitri Van Landuyt, Wouter Joosen
iMinds-DistriNet Research Group, KU Leuven
Celestijnenlaan 200A, 3001 Heverlee, Belgium

{firstname}.{lastname}@cs.kuleuven.be

ABSTRACT

When successful, multi-tenant SaaS applications service many customer organizations (tenants) at once, and SaaS providers face the challenge of complying to the different SLAs of each of these tenants. As a consequence, evolving a SaaS application is in practice done at run time to limit service disruptions, and preferably on a gradual, tenant-per-tenant basis, while taking into account the nature of the upgrade at hand but also these different tenant SLAs.

The economic viability and cost-effectiveness of a SaaS offering depends strongly on two principles: (i) maximal automation of its operation, and (ii) self-service: allowing tenant organizations themselves to customize and configure different aspects of the service to their specific needs.

In this position paper, we highlight the value of adopting the principles of self-adaptive systems in the design of middleware solutions that support continuous evolution of multi-tenant SaaS applications as a means to implement the first principle. Furthermore, we discuss the additional challenges imposed by the second principle, more specifically for supporting tenant mediation, i.e. introducing human stakeholders such as tenant administrators into the inner control-loop of a self-adaptive system.

We present the design of our middleware that addresses these challenges for the specific purpose of evolving multi-tenant SaaS applications, but also discuss the relevance for self-adaptive systems that support stakeholder mediation in general.

1. INTRODUCTION

In the Software-as-a-Service (SaaS) delivery model, business applications are offered at low cost as Internet services by a SaaS provider and remotely consumed by many different customer organizations (called *tenants*). The cost-effectiveness of a SaaS offering is attained by maximally sharing the run-time resources among tenants (a tactic called *multi-tenancy*), which, in turn, allows for large-scale operation and the attainment of economies-of-scale benefits [9]. However, not all tenants are equal, and tenants typically agree upon individual quality terms (SLAs).

Evolving such a multi-tenant SaaS application is challenging as it must respect tenant SLAs even while introducing change to the application. To perform upgrade enactment in a long-living system [3, 23] at run time, mechanisms are required (i) to gradually activate an upgrade in the SaaS application, on a per-tenant basis, keeping in mind tenant-specific SLA stipulations, and (ii) to minimize SLA violations by maintaining fine-grained control over the enactment that accounts for the nature of the upgrade at hand [20].

The underlying principle of self-adaptive systems —autonomously reacting to environmental changes by effecting a corresponding change to the system [8]—, is well suited to support SLA-aware upgrade enactment for the purpose of evolving long-running multi-tenant SaaS applications. The architectural pattern of a feedback-driven control loop [36, 6] is a compelling design strategy for triggering the activation of changes when a certain system state is attained. This reactive behavior facilitates the prioritization of service quality over upgrade enactment to support SLA compliance for the duration of the enactment. Moreover, the autonomous nature of self-adaptive systems allows for fine-grained upgrade enactment mechanisms that remain scalable. Such enactment protocols can take into account many factors such as the overall system state, the specific nature of the upgrade at hand, tenant SLAs, etc. In addition, self-adaptive systems perform self-optimization, and thus can be focused on overall system goals, such as minimizing the risk of SLA violation for the tenants, minimizing the overall cost involved with evolution for the SaaS provider, maximizing the effectiveness of tenant clustering, etc. As a primary contribution of this paper, we present and discuss our middleware architecture that supports such fine-grained run-time upgrade enactment and is structured as a self-adaptive system.

The conditions (e.g. in terms of system load, or system throughput) for the autonomic behavior of fully-automated self-adaptive systems are defined statically and hard-coded in tenant SLAs. However, in scenarios where the SLA cannot be completely fulfilled, additional input from a tenant administrator (a role that acts on behalf of the tenant), might be essential: this administrator has the ability to interpret the current business context of the tenant organization, and therefore grant temporary SLA relaxations.

In practice, this involves introducing a human stakeholder to the inner control loop of the self-adaptive system. This is highly challenging as the tenant administrator is not necessarily aware of other tenants (tenant isolation [27]), the current system state, nor of its inner workings. In addi-

SEAMS'16 May 16–17, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2897053.2897057

tion, the introduction of a non-deterministic, comparably slow human might risk cancelling out some of the benefits of an autonomous system. As a secondary contribution of this position paper, we present and discuss a second version of our middleware that supports *tenant mediation*, i.e. the controlled acquisition of input from a human stakeholder as part of the control loop, input that otherwise would not be accessible to the system.

We discuss and evaluate these mechanisms and their challenges in the context of an example that is based on an industrial SaaS offering, a document processing system.

The remainder of this paper is structured as follows: Section 2 discusses the challenges of evolving multi-tenant SaaS applications in the context of a motivating example. Then, Section 3 presents and discusses our self-adaptive middleware for autonomous and fine-grained tenant-per-tenant upgrade enactment, which is extended with support for tenant mediation in Section 4. Section 5 discusses the challenges of stakeholder mediation in the context of self-adaptive systems, whereas Section 6 discusses related work, and finally, Section 7 concludes the paper.

2. BACKGROUND

This section first introduces the required background concepts in the domain of multi-tenant Software-as-a-Service (SaaS) over a running example. Then, we elaborate on the challenges of continuous evolution and incompatible upgrades.

2.1 Multi-tenant SaaS

The running example for this paper is a real-world multi-tenant Software-as-a-Service application for document processing [21]: After receiving raw data from tenants as input, this application generates batches of documents (e.g. invoices or payslips) for a wide variety of customer organizations (tenants). Tenants can select additional services such as digital signing, and are offered a variety of options for distribution such as printing and sending by regular mail or delivering a PDF via e-mail. Multi-tenant SaaS applications are commonly built as service-oriented architectures for reasons of elastic scalability. Moreover, cost-efficient operation and maintenance are crucial: efforts and resources must be maximally shared among many tenants to attain true economies-of-scale effects (a tactic called *multi-tenancy* [9]).

Different tenants impose different non-functional requirements on the SaaS application. As an example from the document processing application, we consider two tenant organizations, a HR agency and a financial institute. The former outsources the generation and distribution of payslips for and to its employees. Its common workload involves batches of varying size (i.e. amounts of generated documents) and cost (e.g. regular mail vs. e-mail distribution), and is characterized by a peak load at the end of the month but has only limited activity during the remainder of the month. Generating account statements and bills for the financial institute, however, involves small-sized batches and the corresponding workload is more randomly spread over time with less predictable peaks.

A tenant administrator configures the application according to the tenant business' day-to-day requirements in availability (uptime) and performance (in terms of latency or throughput). The HR agency, for example, requires limited document-processing service-capacities during the month, but large capacity at the end of the month (when payslips are issued). The financial institute on the other hand involves less reasonable workloads, and therefore requires more constant provisioning. A shared consistency-related SLA rule is focused on minimizing the management cost and complexity for the tenant associated to cryptographic key management: "*A batch of documents has to be signed with the same cryptographic key material.*".

2.2 Continuous Evolution

As a long-running service, a multi-tenant SaaS application must eventually undergo evolution, and upgrade[1] enactment mechanisms are required that (i) operate at run time, and (ii) maximally respect tenant SLAs. A variety of measures are commonly applied to minimize impact of and during a dynamic enactment of an upgrade [20]: gradual tenant-by-tenant activation of an upgrade, for example, isolates the upgrade enactment of one tenant from the another's [27] (e.g. in terms of workloads). The flipside of the coin, however, is that it causes a reality of many co-existing versions of key components or services of the multi-tenant SaaS application. Accounting for the nature of the upgrade, as an another example, has the potential to additionally lower the impact on the system under upgrade [19]. However, unanticipated changes can be incompatible [5, 1] to the current version of the SaaS application [29], and as such can not be enacted without violating (some) tenant SLAs.

An example upgrade scenario from the context of the document processing SaaS application involves an unanticipated but critical and urgent change to the component that issues digital signatures (a security bug fix), that involve replacing the cryptographic algorithm with a strengthened version and changing the cryptographic keys.

Figure 1 illustrates two alternative transition strategies, $S1$ and $S2$, to enact this example upgrade.

Strategy `S1` (conservative): This transition strategy forces the digital signing component to approach quiescence [25] before, remain idle during, and to resume work after the transition phase (as depicted in Figure 1).

Strategy `S2` (optimistic): As shown in Fig. 1, this strategy involves activating the new version of the digital signing component immediately after deployment.

For document batches that have been partially processed before the transition phase, this strategy will break the SLA rule that all documents in the same batch should be signed with the same key, actually rendering the entire batch inconsistent.

Although, $S1$, the de-facto standard procedure for upgrade enactment, $S1$, ensures application consistency, it causes temporary service disruption and SLA violations. $S2$, the optimistic transition strategy, in turn breaks the consistency constraints.

Motivation. The observations we made from our example SaaS application show that tenants are willing to make such trade-offs, i.e. sacrifice in terms of one service quality, such as consistency, in favor of another, for example availability, but more crucially, (i) that such **trade-offs are specific to tenants**, (ii) that these decisions **are highly specific to the current tenant business context** and (iii) the

[1]In this paper, we focus on upgrade scenarios which are more challenging than update scenarios. However, our findings apply to updates as well.

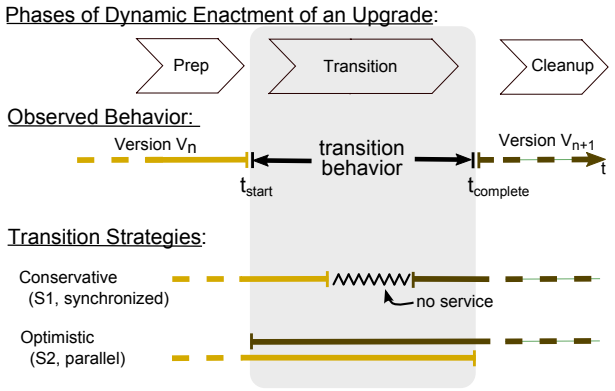

Figure 1: Timeline illustrating two alternative Transition Strategies for the example scenario.

desired resolution strategy can **be highly scoped** (e.g. different trade-offs can be made at level of individual document batches).

For example, if the critical upgrade is rolled out at the end of the month, i.e. during the workload peak of the HR agency, the HR agency may be willing to tolerate inconsistent document batches (prefer $S2$) in favor of complying with their internal deadlines (and then restart the affected batches after the peak-workload period). However, the tenant might prefer to exclude particularly expensive batch jobs (e.g. large amounts of documents that are to be sent by regular mail) from this preference, even if that causes additional delays (hence $S1$).

If the financial institute has released only recently a business product which causes an increased demand for their service that in turn generates document-processing jobs during "rush hours" every day, they might also favor $S2$, i.e. relaxing the batch consistency constraint at the cost of paying twice for invalid batches, but similarly, they might desire to exclude certain jobs.

Problem Statement. In the current state of practice (cf. [30]), the complexity involved in such fine-grained enactment of upgrades is mainly dealt with manually. This however is (i) highly inefficient, (ii) error-prone and imprecise, and (iii) unsustainable as it leads to maintenance nightmares over longer periods of time.

In this position paper, we present a middleware that provides systematic support for the fine-grained, tenant-aware enactment of upgrades that is based on the principles of self-adaptive systems and remains compliant to the key cloud computing principles of maximal automation and self-service.

3. MIDDLEWARE FOR SELF-ADAPTIVE UPGRADE ENACTMENT

Figure 2 presents the high-level design of our middleware architecture that supports fine-grained, tenant-per-tenant upgrade enactment at run time. The top layer in Figure 2 depicts the SaaS application that is (as mentioned earlier) structured as a service-oriented application (i.e. composed of service components) and deployed on top of our middleware.

Run-time Customization Layer. Central to the run-time customization layer (depicted in gray in Figure 2) is the

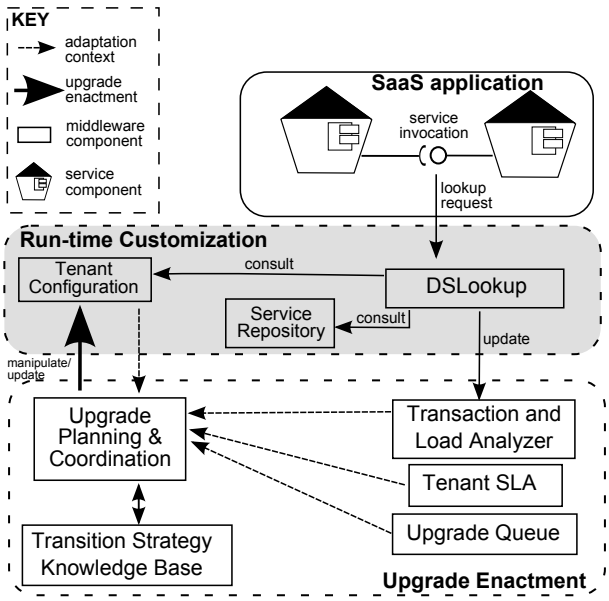

Figure 2: Self-adaptive Middleware for Dynamic Upgrade Enactment.

`DSLookup` (dynamic service lookup) component, which is a middleware component responsible for service lookup (dynamic dispatch) that takes the tenant configuration into account. It is a key design decision in our middleware to enact adaptations at the level of service components, these being the smallest unit of change [26]. By associating the lookup request to (i) the tenant associated to the lookup request, and (ii) the ongoing application transaction[2], the `DSLookup` component performs a tenant- and transaction-specific lookup of the appropriate target service and consults the service repository for available instances of that particular service.

A secondary responsibility of the `DSLookup` component is to update the `Transaction and Load Analyzer` component that represents the monitoring infrastructure keeping track of overall system health and ongoing tenant transactions. By issuing such updates on a service lookup basis, an accurate view is maintained on the overall system.

Upgrade Enactment Layer. The self-adaptive middleware layer for upgrade enactment (the bottom layer in Figure 2) relies on the run-time customization layer to perform gradual, tenant-per-tenant upgrade enactment. More specifically, by manipulating the tenant configurations, for example, reference to an upgraded instead of the previous instance of a specific service is selectively returned.

This manipulation is done by the `Upgrade Planning & Coordination` component that autonomously decides which transition strategy shall be adopted, taking into account a wide range of contextual parameters, such as (i) the upgrade queue (the list of upgrades to be activated per tenant), and (ii) the nature of the upgrade at hand, (iii) tenant SLAs, (iv) the current tenant configuration, (v) overall

[2]To deal with application-specific dynamic dependencies [31] in a generic way, we apply the notion of *application transactions* to group a set of service requests that are subject to the same consistency rules [20]. In our example SaaS application, a batch of documents represents such a transaction.

system health, (vi) ongoing tenant transactions, (vii) ongoing and planned upgrades. The `Upgrade Planning & Coordination` component implements the self-adaptive control loop, continually keeping track of these context parameters and manipulating the tenant configuration accordingly. It performs the longer-term upgrade planning and coordinates change.

Upgrade Planning & Coordination. We now focus on the inner workings of the `Upgrade Planning & Coordination` component, more specifically on how it takes quality considerations into account (tenant SLAs) in its decision logic. This component implements a decision model that calculates for each potential transition strategy the associated rewards and penalties, e.g. incurred by complying or disobeying specific SLA rules, and picks the strategy that has the optimal net reward (rewards minus penalties).

These penalties and rewards are highly specific to the observed context; i.e. to the tenant (based on the tenant SLA), to tenant transactions (the cost to cancel ongoing transactions depends on the current progress and the cost to restart transactions), to the upgrade at hand (e.g. its compatibility, its urgency), to the current system state (e.g. current and predicted load), etc.

To accomplish this, these contextual parameters are collected and converted into elements of the meta-model presented in Figure 3. These `Quality Implication` elements express the impact of certain contextual elements (`Scope` tree) on the qualities of interest (`Quality` tree).

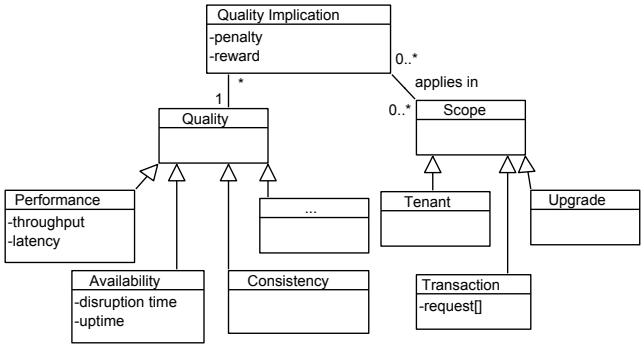

Figure 3: System quality meta-model.

A transition strategy (cf. adaptation strategy [17]) implements a particular protocol for upgrade enactment. Such a protocol is co-developed with the software upgrade and defines a sequence of concrete changes [20]. The knowledge about these applicable transition strategies are centralized in the `Transition Strategy Knowledge Base`. Moreover, as shown in Figure 4, the quality implications of picking a specific strategy (e.g. inherent trade-offs) are calculated.

In the running example, transition strategy $S1$ prioritizes application consistency over availability while transition strategy $S2$ prioritizes availability (service continuity) over consistency. This crucial information is encoded in the metadata of each available transition strategy.

As mentioned in Section 1, an autonomous system is self-optimizing towards overall system goals. Relevant system goals in the context of our middleware are: minimization of SLA violation, minimize cost involved, reduce the total upgrade time for all tenants, etc. Based on the results of

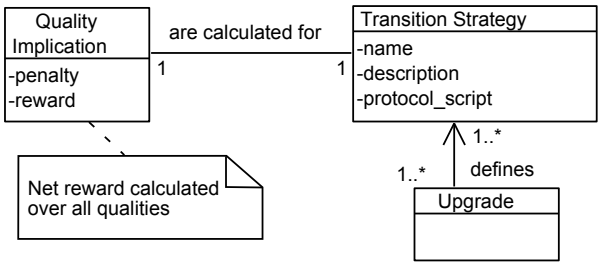

Figure 4: Knowledge meta-model for transition strategies.

gathering and calculating the quality implications for each strategy, the transition strategy resulting in best overall value can now be selected autonomously.

The relatively straightforward penalty-reward scheme presented here for illustrative purposes is a form of *change impact analysis* [2], i.e. predicting the impact of specific maintenance activities and updates on the system, but evidently more sophisticated algorithms may be considered.

There are two key benefits to performing upgrade enactment in a self-adaptive control loop. One, the enactment activities, which consume additional resources in the best case and may impact the system's SLA in the worst case, are given less priority than the system's primary objective to service within quality boundaries. Two, the autonomous nature of a self-adaptive system renders even complex enactment protocols scalable and maximally automated, and allows for sophisticated enactment protocols that involve fine-grained tenant context-aware transition behaviors for the enactment of an upgrade.

4. SUPPORT FOR TENANT MEDIATION

A middleware for autonomous upgrade enactment (such as the middleware presented in Section 3) encounters its limits when it comes to upgrades that necessarily violate tenant-SLAs (cf. incompatible updates [5]), such as the example upgrade scenario described in Section 2.2 (an urgent security bug fix).

A successful mitigation strategy in such exceptional situations, is to temporarily relax tenant SLAs [20]. Tenant SLAs are statically defined and specify service quality constraints that are enforced during regular operation. They do not allow taking into account the current business context of the tenant and thus do not facilitate the definition of suitable SLA relaxations. This essentially requires the otherwise autonomous self-adaptive enactment system to interact with human stakeholders, which we call *tenant mediation*. This section extends our middleware with such support.

Adding mediation capabilities to a self-adaptive control loop is highly challenging for two reasons: (R1) it requires tight integration between the inner control loop the self-adaptive system and the self-service interface (the dashboard offered to the tenant administrator), (R2) there is a knowledge and abstraction gap: the tenant administrator is typically not aware of the internals of the SaaS application nor of the details of the upgrade at hand and may only be able to exercise control over the enactment in terms of high-level business- or even domain-specific abstractions.

Figure 5 depicts our initial solutions to both key requirements, which we outline in the remainder of this section.

In Section 5 we provide an in-depth account of remaining challenges in this context.

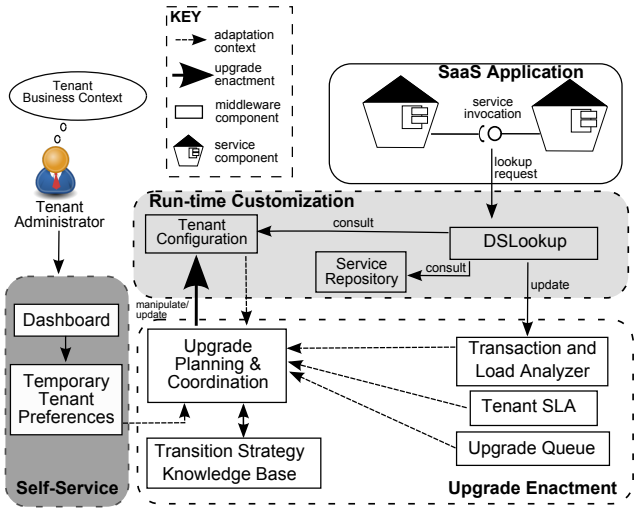

Figure 5: Self-adaptive Middleware for Dynamic Upgrade Enactment with support for Tenant Mediation.

Integrating Tenant Self-Service Interfaces. To address R1, we integrate the existing tenant dashboard and self-service systems into the autonomous control loop described above. In Figure 5, this is accomplished by allowing the tenant administrator to insert *temporary quality preferences* (typically, SLA relaxations) as another context element that is taken into account by the `Upgrade Planning & Coordination` component which implements the self-adaptive control loop. When the tenant administrator has not provided any temporary quality preferences in the time window exactly defined by the SaaS provider, the middleware will behave as the automated self-adaptive upgrade enactment presented in Section 3.

These temporary SLA relaxations are based on the tenant administrator's knowledge of the current business context (i.e. ongoing business transactions, predicted peaks in orders, etc) and provide the self-adaptive systems with the means to temporarily disobey the constraints set in the tenant SLA for the purpose of upgrading the system.

From the point of view of the tenant, such a mechanism provides him with a way to exert control on how unavoidable incompatible upgrades are enacted for them, on the trade-offs involved and on aspects such as timing, as opposed to a situation in which decisions would be forced onto them by the SaaS provider.

Bridging the Gap. Figure 6 presents our meta-model for expressing temporary quality preferences, which is how the tenant administrator provides his inputs (R2) to the self-adaptive system. As shown, such temporary quality preferences are defined for a scope that is at least limited in time, but the tenant administrator can also scope the temporary preference in terms of for example, the transactions involved (as shown in Figure 3).

As this abstraction of quality preferences is a close match to the abstractions presented in Figure 3 related to qualities, the upgrade enactment layer can easily be extended with support to take these additional contextual parameters into account. This is specifically done with the reward-penalty system, i.e. by granting stronger rewards to transition strategies that are compliant to temporary quality preferences.

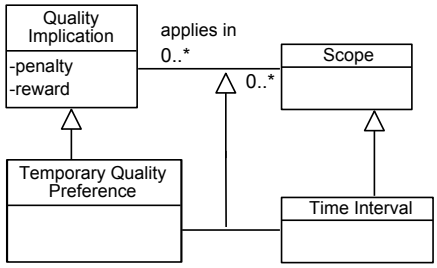

Figure 6: Meta-model for specifying temporary quality preferences.

5. DISCUSSION

This section covers a number of aspects specific to the self-adaptive middleware presented above and summarizes our analysis of the benefits and challenges of human stakeholder mediation in self-adaptive systems in general.

MAPE-k. Aligning our middleware to the widely-known MAPE-k reference architecture for self-adaptive systems [24] is relatively straightforward: the different sources of context information (`Upgrade queue`, `Tenant SLA`s, `Transaction and Load Analyzer` and the `Tenant configurations`) implement the role of *Monitoring* the environment or system context. The `Upgrade Planning and Coordination` component in turn implements the roles of *Analysis* and *Planning* (as it includes maintaining and enacting the upgrade schedule). Finally, the actual *Enactment* is done by our runtime customization layer, more specifically by updating the tenant configuration and leveraging the dynamic nature of the `DSLookup` component.

The extended middleware in Section 4 effectively introduces a human stakeholder (the tenant administrator) in the role of an additional *monitor* in the system. More specifically, the tenant administrator in fact provides his understanding of the tenant's business context as an input to the self-adaptive upgrade-enactment middleware.

Stakeholder Mediation in Self-Adaptive Systems: remaining challenges. It may seem from the initial architecture presented in this paper that integrating a human stakeholder in a fully autonomous self-adaptive system is relatively straightforward, by virtue of design decisions related to modular design and introducing the human as another monitor or information provider. It is however essential to note that this is in fact an underestimation of the involved complexity. More specifically, to make such a scheme effective, tighter integration will be required between the human stakeholder[3] and:

Analysis subsystems. For a human stakeholder to make effective and well-informed decisions, the system would

[3]or its dashboard components by which he or she is represented in the system.

have to (partially) share some results of its (change impact) analysis activities, and communicate to the tenant administrator more directly the exact nature of the trade-offs involved and the (predicted) consequences of the temporary SLA relaxations. As mentioned, this requires tighter integration between the monitoring and analysis subsystems. In addition, the mentioned abstraction and information gap (R2) remains an unsolved challenge, as providing such analysis outcomes at the appropriate level of detail without disclosing system internals or information about other tenants (as such maintaining *tenant isolation* [27]) becomes highly challenging. This is especially true because the outcomes of these activities in many cases are strongly influenced by factors outside of the tenant scope: e.g. the SaaS provider does not want to explain to the tenant administrator that his upgrade has been postponed because of other ongoing upgrades for other tenants, or because the SaaS application is currently underprovisioned (e.g. low on resources, and in the process of scaling out or up).

Planning subsystems. Furthermore, the human stakeholder is given no further insight nor control on the final outcome of the analysis activities, the concrete upgrade plan that is central to the self-adaptive system and has a large impact on the service quality that is perceived by the human stakeholder. Especially in the context of cloud computing, in which many surveys [33] mention the loss or lack of control as one of the most prominent hurdles against adoption, looking for means to provide more advanced control on such an autonomous control system presents a promising track for future research.

Dustdar et al. [12] mention the limited understanding of the interdependencies between social and technical entities as one of the major challenges in self-adaptive service systems. The problem addressed in this paper effectively introduces social entities (human stakeholders) to the system, but the problems related to not leveraging the interdependencies between the social space (e.g. lack of formalized business context) and cyber space remain unsolved.

6. RELATED WORK

Self-Adaptation Concepts. Although many self-adaptive systems [12, 39] are based on a closed (fully-automated) MAPE-k feedback loop [24], human participation in the control loop has been considered early on, for example, to perform or approve the planning step [34], or to support enactment [7, 15]. To the best of our knowledge, human actors have not been utilized as an additional "sensor" in an otherwise automated feedback loop, and to introduce awareness of the social space [12] into the self-adaptive system.

In policy-based approaches [18, 13], a policy often consists of a pre-defined set of event-conditions-action rules, and controls the planning step which automatically chooses one among a set of alternative adaptation strategies [17, 34]. Others [4, 35] specify desirable target conditions for the system, for example in a goal model –which may even be evolved at run time [34, 32, 37]–, and the activities to reach a target condition are computationally reasoned over. However, our self-adaptation middleware is based on *dynamic* tenant preferences (run-time, fine-grained and temporary exceptions to otherwise static SLA policies) that are taken into consideration in addition to an autonomously-reasoning closed feedback loop.

SLA Negotiations. Automated negotiation of SLAs between a service consumer and provider has been explored from a strategic (i.e. how to steer its outcome to one participants's interest) and platform perspective [41, 40]. In contrast, our approach deals with *scoped* tenant SLAs of *temporary* effect and contexts of *different abstractions*, and must relies on a fall-back option when the tenant does not provide temporary quality preferences in time.

Dynamic Upgrade Enactment. Enacting a software upgrade at run time has been studied extensively for decades, resulting in the dominant approaches to enact either (i) forward-compatible changes without affecting consistency or availability [22] or (ii) incompatible changes focusing on (iia) consistency [25, 31] or on (iib) availability [1]. Our approach supports the utilization of more than one protocol to enact an upgrade in order to support customization of the transition behavior of the system to the requirements of its users (tenant organizations).

Evolution of Multi-tenant SaaS application. Middleware and other frameworks to upgrade multi-tenant SaaS applications at run time either promote consistency at all cost [11], are applicable to anticipated upgrades only [28, 16], consider special-purpose applications only [14], or rely on pro-active enactment of dynamic upgrades [19]. Run-time tenant-driven customization of Multi-tenant SaaS offerings remains a challenging research topic [38].

7. CONCLUSION

Tenant organizations outsource parts of their day-to-day workload to multi-tenant SaaS applications, and thus strongly depend on qualities such as service continuity and high availability. However, as any other long-running service, SaaS applications must eventually evolve, which in case of incompatible upgrades leads to a challenging situation for the SaaS provider in which they cannot entirely fulfill their obligations specified in tenant SLAs.

In this position paper, we introduced and discussed our self-adaptive middleware that allows tenant administrators to relax their SLAs temporarily by specifying temporary quality preferences, a mechanism which we termed *tenant mediation*.

The influential and visionary article entitled "*Software engineering for self-adaptive systems: A second research roadmap*" [10] discusses the discrepancy between software evolution and maintenance processes that are traditionally both human stakeholder-driven and of a discrete nature, and the scope of self-adaptive systems which are more suited for enacting continuous change. In addition, the authors state that marrying both worlds will require fundamental changes to traditional Software Engineering processes.

This position paper provides a concretization of these visionary ideas in the specific context of multi-tenant SaaS applications and cloud computing, domains which currently play a catalytic role in modernizing and rethinking traditional development processes, under the banner of keywords such as Continuous Development, Continuous Integration, and DevOps.

Acknowledgments. This research is partially funded by the Research Fund KU Leuven, the ADDIS research program funded by KU Leuven GOA, and the DeCoMAdS SBO strategic research project.

8. REFERENCES

[1] S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In *ECOOP*. 2006.

[2] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: A roadmap. In *ICSE*, 2000.

[3] J. Bosch and R. Capilla. Dynamic variability in software-intensive embedded system families. *Computer*, 2012.

[4] V. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, and S. Uchitel. Morph: A reference architecture for configuration and behaviour self-adaptation. *arXiv preprint arXiv:1504.08339*, 2015.

[5] E. Brewer. Lessons from giant-scale services. *Internet Computing, IEEE*, 5(4):46–55, Jul 2001.

[6] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. *Software Engineering for Self-Adaptive Systems*, chapter Engineering Self-Adaptive Systems through Feedback Loops, pages 48–70. 2009.

[7] J. Cámara, G. A. Moreno, and D. Garlan. Reasoning about human participation in self-adaptive systems. In *SEAMS*, 2015.

[8] B. Cheng et al. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*. 2009.

[9] F. Chong and G. Carraro. Architectural strategies for catching the long tail., http://msdn.microsoft.com/en-us/library/aa479069.aspx, 2006.

[10] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.

[11] T. Dumitraş and P. Narasimhan. Why do upgrades fail and what can we do about it?: Toward dependable, online upgrades in enterprise system. In *Middleware*, 2009.

[12] S. Dustdar, C. Dorn, F. Li, L. Baresi, G. Cabri, C. Pautasso, and F. Zambonelli. A roadmap towards sustainable self-aware service systems. In *SEAMS*. ACM, 2010.

[13] A. Erradi, P. Maheshwari, and V. Tosic. Policy-driven middleware for self-adaptation of web services compositions. In *Middleware*, 2006.

[14] S. Ertel and P. Felber. A framework for the dynamic evolution of highly-available dataflow programs. In *Middleware*, 2014.

[15] D. Eskins and W. Sanders. The multiple-asymmetric-utility system model: A framework for modeling cyber-human systems. In *QEST*, 2011.

[16] J. García-Galán, L. Pasquale, P. Trinidad, and A. Ruiz-Cortés. User-centric adaptation of multi-tenant services: Preference-based analysis for service reconfiguration. In *SEAMS*, 2014.

[17] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 2004.

[18] J. C. Georgas and R. N. Taylor. Policy-based self-adaptive architectures: A feasibility study in the robotics domain. In *SEAMS*. ACM, 2008.

[19] F. Gey, D. Van Landuyt, and W. Joosen. Middleware for customizable multi-staged dynamic upgrades of multi-tenant saas applications. In *UCC*, 2015.

[20] F. Gey, D. Van Landuyt, W. Joosen, and V. Jonckers. Continuous evolution of multi-tenant saas applications: A customizable dynamic adaptation approach. In *PESOS*, May 2015.

[21] F. Gey, S. Walraven, D. Van Landuyt, and W. Joosen. Building a customizable Business-Process-as-a-Service application with current state-of-practice. In *Software Composition*, 2013.

[22] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster. Specifying and verifying the correctness of dynamic software updates. In *Verified Software*. 2012.

[23] C. Hofmeister and J. Purtilo. Dynamic reconfiguration in distributed systems: adapting software modules for replacement. In *Distributed Computing Systems*, 1993.

[24] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan 2003.

[25] J. Kramer and J. Magee. The evolving philosophers problem: dynamic change management. *Software Engineering*, 1990.

[26] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE*, 2007.

[27] R. Krebs, C. Momm, and S. Kounev. Metrics and techniques for quantifying performance isolation in cloud environments. *Science of Computer Programming*, 90, Part B:116 – 134, 2014. Special Issue on Component-Based Software Engineering and Software Architecture.

[28] I. Kumara, J. Han, A. Colman, and M. Kapuruge. Runtime evolution of service-based multi-tenant saas applications. In *ICSOC*. 2013.

[29] M. Lehman, J. Ramil, P. D. Wernick, D. Perry, and W. Turski. Metrics and laws of software evolution-the nineties view. 1997.

[30] T. Limoncelli, S. Chalup, and C. Hogan. Upgrading live services. In *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems*, chapter 11. Pearson Education, 2014.

[31] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna, and J. Lu. Version-consistent dynamic reconfiguration of component-based distributed systems. In *FOSE*, 2011.

[32] B. Morin, O. Barais, J. Jezequel, F. Fleurey, and A. Solberg. Models@run.time to support dynamic adaptation. *Computer*, 42, 2009.

[33] B. Narasimhan and R. Nichols. State of cloud applications and platforms: The cloud adopters' view. *Computer*, (3):24–28, 2011.

[34] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems*, 1999.

[35] L. Sabatucci and M. Cossentino. From means-end analysis to proactive means-end reasoning. In *SEAMS*. IEEE, 2015.

[36] M. Shaw. Beyond objects: A software design paradigm based on process control. *Softw. Eng. Notes*, 1995.

[37] V. E. S. Souza, A. Lapouchnian, and J. Mylopoulos. (requirement) evolution requirements for adaptive systems. In *SEAMS*. IEEE, 2012.

[38] D. Van Landuyt, S. Walraven, and W. Joosen. Variability middleware for multi-tenant saas applications. In *SPLC*, 2015.

[39] D. Weyns, M. U. Iftikhar, S. Malek, and J. Andersson. Claims and supporting evidence for self-adaptive systems: A literature study. In *SEAMS*. IEEE, 2012.

[40] E. Yaqub, P. Wieder, C. Kotsokalis, V. Mazza, L. Pasquale, J. L. Rueda, S. G. Gómez, and A. E. Chimeno. A generic platform for conducting sla negotiations. In P. Wieder, M. J. Butler, W. Theilmann, and R. Yahyapour, editors, *Service Level Agreements for Cloud Computing*, 2011.

[41] X. Zheng, P. Martin, K. Brohman, and L. D. Xu. Cloud service negotiation in internet of things environment: A mixed approach. *IEEE Transactions on Industrial Informatics*, 10(2):1506–1515, May 2014.